WebNN implementation in Chromium

9/17/2021

Overview

As we know, the deep learning (DL) workload has been deployed to web with emerging JavaScript framework, e.g. Google's TensorFlow.js / TenSorFlow-Lite Web and Microsoft's ONNX.js / ONNXRuntime Web, OpenCV.js. These frameworks utilize WebAssembly API to compute on CPU and WebGL / WebGPU API to compute on GPU, but can't access the platform capabilities beneficial for ML with dedicated ML hardware accelerators. So Web Neural Network (WebNN) was incubated, which is a new web standard API for accessing hardware accelerators from web browsers, WebNN defines the JavaScript API that allows Web applications to build the neural network computation graphs and execute the graphs with hardware acceleration. So these frameworks can improve inference efficiency by using WebNN to access hardware accelerators.

The WebNN API is being standardized by the W3C Web Machine Learning Working Group (WebML WG) after a two-year incubation period in the Community Group. The Working Group has published a First Public Working Draft of the WebNN API and plans to release the Candidate Recommendation in Q2 2022. The WebNN API can be implemented in Chromium browser by using the available native operating system machine learning APIs such as Android / ChromOS Neural Network API, Windows DirectML API and macOS/iOS ML Compute API, these native API will talk with drivers to run WebNN primitives on various machine learning hardware including CPU, GPU and dedicated Machine Learning hardware accelerators.

Summary

Propose to implement WebNN standard API in Chromium browser to improve JavaScript inference efficiency by using the available native operating system machine learning APIs. There are three approaches to implement WebNN in Chromium:

1. Implement CPU backend in render process
2. Implement WebNN GPU backend in the GPU process with a service which communicates with the render process with mojo infrastructure.
3. WebNN-native is one native implementation of WebNN API, the infrastructure is reused with Dawn including code generation tool, validation and error handling. The implementation is to use WebNN-native to implement WebNN API in GPU process for GPU backend and use Command Buffer for Inter-process-communication.

Platforms

Mac, Windows, Linux, Chrome OS, Android, Android WebView, iOS.

Contributors

Junwei Fu (junwei.fu@intel.com), Ningxin Hu (ningxin.hu@intel.com) (WebNN Spec editor), Belem Zhang (belem.zhang@intel.com), , Jiawei Shao (jiawei.shao@intel.com) (WebGPU contributor)

Reviewers

Corention Wallez (WebGPU chair)

Bug

There is an overall issue filed in the WebNN native repository.

Code affected

Implement WebNN WebIDL in blink/modules/webnn, and add command buffer for WebNN in gpu/command_buffer, Implement ContextProvider that provides a WebNN implementation over command buffer to GPU process in services/viz/public/cpp/gpu/context_provider_command_buffer.cc.

Design

High Level overview

Figure 1 illustrates where WebNN API fits in the software stack. It defines a set of common building blocks including constant trained values, base operations such as convolution, pooling, activation. For other operations which WebNN does not support will be delegated to WebAssembly or WebGPU. By using WebNN primitives, the JavaScript machine learning framework can define a computational graph which represents part or whole of a machine learning inference model. Using WebNN API to compile and execute the graph with hardware acceleration, of course, the execution of the WebNN graph can interact with kernels written in WebAssembly or WebGPU compute shader. With that, the frameworks can be flexible by using the WebNN for hardware acceleration and using WebAssembly, WebGPU for other operations support.

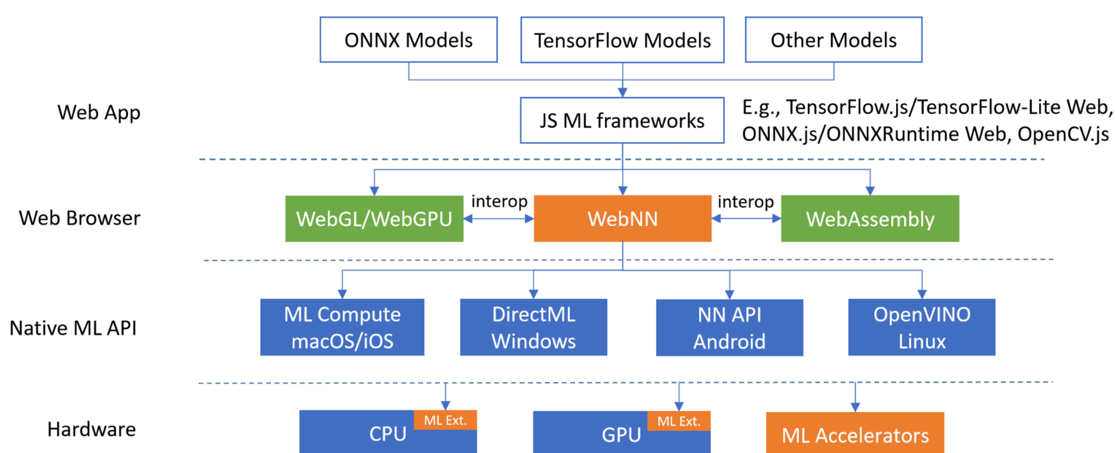
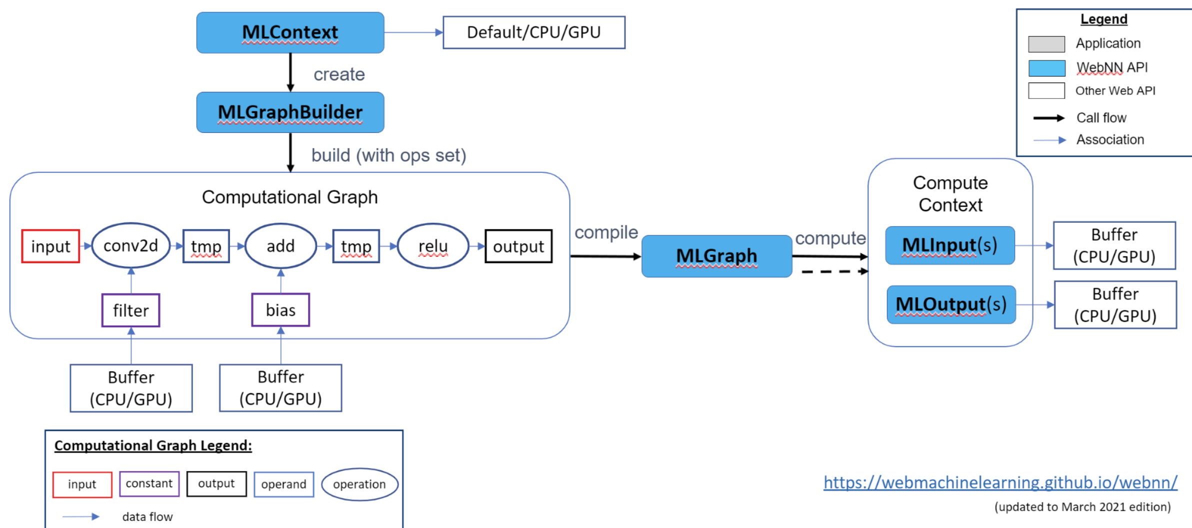


Figure 1 WebNN in software stack

So Web Browsers need to implement the WebNN API using native machine learning API available in the operating system. The primitives of WebNN can be mapped to the native machine learning API available on different operating systems, such as Android Neural Network API, DirectML on Windows, ML Compute on macOS/iOS, and OpenVINO on Linux. Eventually, these native APIs will talk with compilers and drivers to run these primitives on various machine learning hardwares CPU, GPU and dedicated Machine Learning hardware. For the deep learning frameworks or libraries which are not embedded in the OSes, such as OpenVINO, we need to compile and install them for use in advance. Otherwise, WebNN will throw an error if the user wants to use the corresponding framework to accelerate.

WebNN API workflow

We have implemented the WebNN API proposal in JavaScript WebNN-polyfill, which can be understood as an emulated implementation of proposed functionality using pre-existing mechanisms. Polyfills are commonly used by the Web community and standards organizations to understand and illustrate proposed APIs.



In the WebNN API, the MLContext object represents a global state of neural network execution, the MLGraphBuilder defines a set of operations to build a computational graph, such operations may be accelerated with dedicated hardware such as the GPUs, CPUs with extensions for deep learning, or dedicated ML accelerators. The WebNN API provides interfaces to build a computational graph, compile the graph and execute the graph, the compute function in MLGraph provides inputs/outputs to interact with other frameworks.

Implement CPU backend in Render process

WebNN CPU backend depends on XNNPACK library and runs in the Renderer process. The MLGraphXnnpack inherits the MLGraph interface. It implements the graph building and computation methods by using XNNPACK APIs for both synchronous and asynchronous execution modes.

All MLGraphXnnpack instances share a cross-threads reference-counted SharedXnnpackContext class that manages the initialization and deinitialization of XNNPACK library and potential pthreadpool. The memory allocation of XNNPACK library would be intercepted to PartitionAlloc In Chromium build. The SIMD buffer would be allocated in the aligned partition. The pthreadpool needs to be integrated into base::ThreadPool. The design is to be done.

XNNPACK Subgraph is an abstract representation of a neural network model. The MLGraphXnnpack graph building method creates the XNNPACK Subgraph object and defines XNNPACK Values for MLGraph's operands and Nodes for MLGraph's operators.

XNNPACK Runtime is a combination of an execution plan for Subgraph Nodes and a memory manager for Subgraph Values. MLGraphXnnpack creates a XNNPACK Runtime object from this Subgraph object. The XNNPACK Runtime object is kept within MLGraphXnnpack for graph computation.

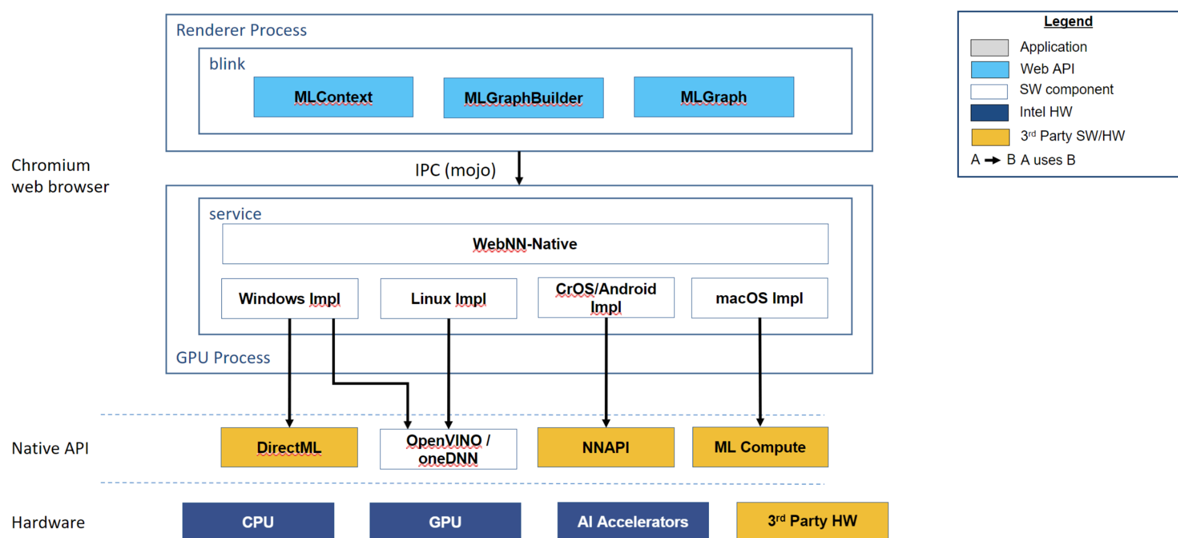
For the MLGraph compute method, MLGraphXnnpack firstly sets up the data pointers of inputs and outputs of the XNNPACK Runtime object to the user-supplied array buffers. Then

MLGraphXnnpack invokes the XNNPACK Runtime that executes the forward pass for all operators in the runtime. After the execution, the results are available in the output array buffers.

For synchronous execution mode, MLGraphXnnpack executes the XNNPACK APIs in the caller's thread. For asynchronous execution mode, MLGraphXnnpack executes the XNNPACK APIs in a background worker thread that avoids blocking the main thread.

Implement GPU backend using mojo for IPC

For security and stability, modern web engines usually employ multi-process architecture that isolate web content (HTML/JavaScript/CSS) execution in sandboxed processes and access platform resources and devices in privileged processes. The rendering engine and JavaScript engine run in the unprivileged process hardened by sandbox and rely on privileged processes to access hardware acceleration. We extended the Chromium rendering engine, "Blink" which runs inside a renderer process, to expose the WebNN JavaScript API. The WebNN JavaScript API was proxied to gain privileged access to the GPU process so that it could make use of hardware acceleration. We used shared-memory inter-process-communication (IPC) to improve the efficiency of transferring (pre-)trained data, inputs, and outputs.



Please get a more detailed design from WebNN implementation with mojo.

Implement DirectML backend on Windows

https://docs.google.com/document/d/1TMs36IE9wL9rNuh8lriGr51S8MU1JezU2SCZy-YqJ3Y/edit?usp=sharing

Implement MLService backend on Chrome OS

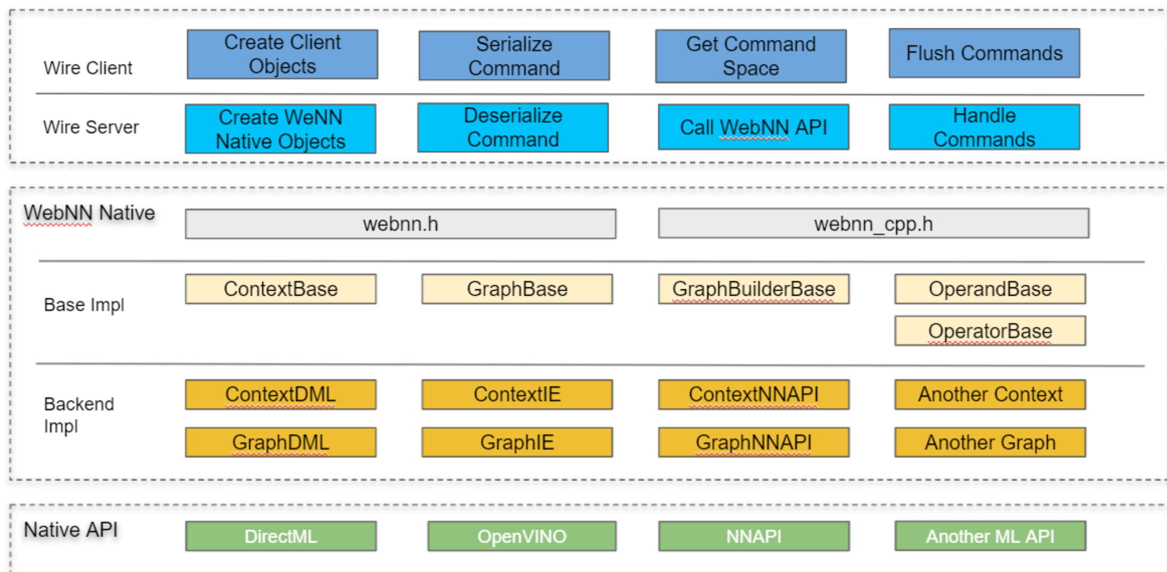
https://docs.google.com/document/d/1KzuWAhQCXATiD_h3HTuCm-nuGg0mooMMZvalLnJ152M/edit?usp=sharing

Implement GPU backend using Command Buffer for IPC

WebNN-native is one native implementation of the Web Neural Network API, the infrastructure is reused with Dawn including code generator tool, validation and error handling. The code generation tool is to generate WebNN interface definition and Wire Layer, there are some files that are reused from Dawn and some files are copied and modified from Dawn.

By reusing Dawn's infrastructure:

1. WebNN can interoperate easily with WebGPU to shared GPU buffer
2. There are the same security mechanism to access GPU device
3. It's easy to develop because some code are generated by tools from webnn. json.
4. It's easy to integrate into Chromium / framework as a component.



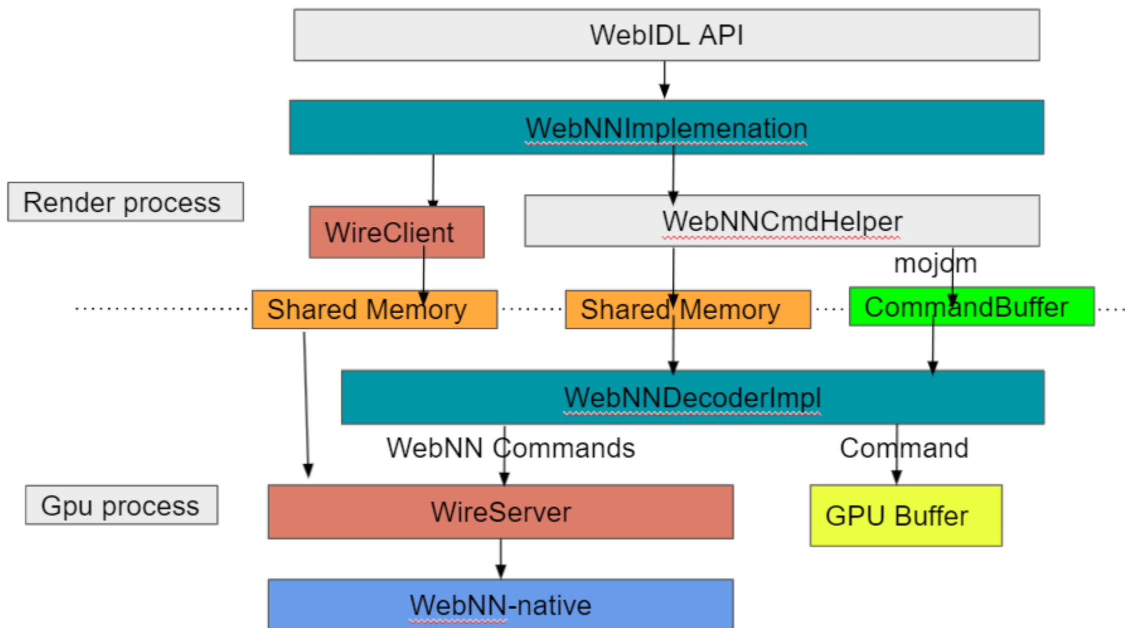
The building blocks are:

- WebNN C/C++ headers that applications and other building blocks use.
    - The Webnn.h is a one-to-one mapping with the WebNN IDL.
    - webnn_cpp.h isa wrapper of C pointer as handle to expose C++ object that will inherit ml::ObjectBase, but it doesn't hold reference count, it will add/decrease

reference count with C API when calling Derived::MLReference(mHandle) /Derived::MLRelease(mHandle)

- Backend Implementations that use platform's ML APIs
- A client-server implementation of WebNN for applications that are in a sandbox without access to native drivers

The WebNN-native implementation inside the GPU process has multiple backends for different host OS API/framework combinations including: (1) NNAPI for Android; (2) ML Compute for macOS; (3) DirectML for Windows; (4) OpenVINO/oneDNN for Linux.



Life of a WebNN call in Chrome in simple terms:
webnn.h->WebNNImplementation->WebNNCmdHelper...SharedMemory...->WebNNDecoderImpl->WebNN-native->DirectML/OpenVINO/Android NN / MLCompute

CommandBuffer is responsible for coordinating communication between WebNNCmdHelper and WebNNDecoderImpl. It has methods for creating and deleting shared memory as well as communicating the current state back and forth. Specifically sending the latest 'put' pointer from the client through AsyncFlush() or Flush() and for getting the latest 'get' pointer through the results of 'Flush'

Please get a more detailed design from [WebNN implementation with command buffer](#).

Rollout plan

Core principle considerations

Security Considerations

This section is referring to Partial WebGPU security doc

Arbitrary user inputs

WebNN exposes three new types of user-input that directly get forwarded to native code.

Risk: WebNN commands in JavaScript will map one-to-one with WebNN/Dawn-native function calls in the GPU process. This means that all commands must be checked for validity and well-formedness on the GPU process side.
Mitigation: The WebNN specification will have strict validation rules that will be tested in the CTS, and the frontend will be fuzzed to discover code paths the test suite might have missed. Well-formedness will be guaranteed by WebNN/Dawn-wire that will produce errors on malformed commands. This part will be fuzzed too.

Risk: WebNN graph (model) will be created by JavaScript by calling WebNN commands (GraphBuilder) and consumed by the graph validator/translator/transforms that are in the GPU process. Everything must be checked on the GPU process side.
Mitigation: all graphs will be validated before they are passed to the translator or transforms. All these components will be fuzzed and the WebGPU CTS will have extensive shader testing.

Risk: arbitrary data uploaded to the GPU (and even AI accelerator) by the application to populate buffers. This data will be given as an ArrayBuffer so it can contain anything, such as values that would cause OOB accesses in GPU operations, or non-finite floats. The same issue arises with arbitrary data being generated on the GPU.
Mitigation: the GPU doesn't care about exact values, and non-finite floats will just produce bad results. OOB indices are in the "GPU Risks" class and treated below.

Architectural risks

The way WebNN is implemented in Chromium might introduce some architectural risks.

Risk: use of shared-memory for passing data and WebNN commands around. Using shared memory is important for speed but will let the renderer process modify data while the GPU process is using it (TOCTOU).
Mitigation: depends on the type of data.

- Data uploaded to the GPU: values don't matter so there is no TOCTOU possible.
- graph (model): compiling graph is expensive, but the graph would be constructed in the server side, so probably there is no TOCTOU issue.
- WebNN commands: they need to be fast, so simple commands (no pointers except to WebNN objects) will be parsed in-place from a volatile pointer to shmem, then validated and executed. More complex commands (e.g., graph build and compute) will be parsed in a temporary buffer.

Risk: performance as a feature. One of the selling points of WebNN is that it will be faster for ML graph compute. This means will need to do some low-level optimizations (e.g., operation fusion etc.,) that are more difficult to reason about and could cause bugs (including security bugs).
Mitigation: use good judgement, have these optimizations separable for unit-testing.

Risk: additional attack surface for the GPU process. WebNN will likely add thousands of lines of code to the GPU process as well as some new GPU driver shared libraries. Exploitation of the GPU process gives access to all the GPU data of other pages and of the browser's chrome.
Mitigation: none, or everything in this document depending on how you see it. WebNN will be simpler to validate than WebGPU and GPU drivers are already present for other APIs so hopefully the attack surface increase isn't that big.

Risk: GPU/AI accelerator drivers are buggy. They are often fine-tuned for popular ML frameworks (e.g., ONNXRuntime and TensorFlow) and benchmarks and other applications have to adapt to their bugs. It is possible to imagine that a driver bug could cause more than just bad results and leak data that shouldn't be accessible.
Mitigation: we will push GPU/AI accelerator vendors to integrate the WebNN CTS as part of their driver testing process, like we did for WebGPU. WebNN/Dawn will implement workarounds for bugs if at all possible, and otherwise blacklist hardware-accelerated WebNN on specific driver versions.

Spectre

WebNN may use the software fallback (such as software backend of DirectML, or even a CPU backend), however, given the user graph is less arbitrary and fully validated, the spectre risk is likely minimum.

Out-of-bounds resource access

"Graphs" running on the GPU/AI accelerator access memory through "resources" that are "buffers" of untyped linear memory or "textures" (when context is created from a GPU device) containing typed 1D/2D/3D memory. Resources are bound to a graph execution by the WebNN implementation; their contents can be accessed by operations of the graph. Because each operation is defined by WebNN spec, the implementation could validate the resources before the graph execution. This would avoid the memory access with a user defined dynamic offset, thus out-of-bounds resource access risk is likely minimum.

Undefined behavior

WebNN targets native OS ML API (such as DirectML) to build and compute a computational graph. The operations defined by WebNN spec would be mapped and tested on native OS ML API. Given the fixed set of functionalities, the risk of undefined behavior is likely minimum. (Need Chai's inputs here)

High precision timers

WebNN spec doesn't provide a way to measure how long the execution of a graph. We should re-assess this risk once WebNN spec provides high-precision timing data.

Overflow validation

Chromium has provided some security tools such as the [base/numerics]() for developers which can avoid overflow errors and so on.

This directory contains a dependency-free, header-only library of templates providing well-defined semantics for safely and performantly handling a variety of numeric operations, including most common arithmetic operations and conversions.

Abuse

This section discusses how the WebNN API could be used for bad things.

Fingerprinting

The root object for WebNN is MLContext, which represents an open connection to one of the compute device, such as GPU or an AI accelerator. Because some computers have two GPUs, we want applications to be able to choose which one to use: the power-efficient or the fast one.

WebNN doesn't expose device information like device names, driver versions and memory size etc.,.

The graph execution timings could be fingerprintable and the precision of results could be fingerprintable. This issue is being raised to WG for discussion: https://github.com/webmachinelearning/webnn/issues/85

Denial-of-service

An application will be able to use WebNN to acquire system resources, such as GPU processing time and GPU memory. We can control how much memory an application uses. And we can calculate the FLOPS of a graph by examining the operations and their parameters (e.g., weights), so we can estimate the GPU processing time before execution.

Cryptomining

It is unclear how WebNN can be used for crypto mining.

Privacy considerations

Testing plan

Followup work